Enumeration types

The keyword `enum` is used to declare enumeration types. It is useful in assigning names to integral constants. Consider, for example, the declaration

```
enum weekday {sun, mon, tue, wed, thu, fri, sat};
```

This creates the user-defined type `enum weekday`. The keyword `enum` is followed by the tag name `weekday`. The enumerators are the identifiers `sun`, `mon`, $\cdots$, `sat`. They are constants of type `int`. By default, the first enumerator is 0, and each succeeding one has the next integer value. The above declaration is an example of a type specifier, which we also think of as a template. No variables of type `enum weekday` have been declared yet. To do so, we can now write

```
enum weekday d1,d2;
```

This declares `d1` and `d2` to be of type `enum weekday`. They can take on as values only the elements (enumerators) in the set. Here is a short code illustrating this.

```
#include <stdio.h>
enum weekday {sun, mon, tue, wed, thu, fri, sat};
int main()
{
 enum weekday d1,d2;
 d1=mon;
 d2=thu;
 printf("%d %d\n",d1,d2);
 return 0;
}
```

Listing 73: C Program "senu.c"

As expected, the output of the code is `1 4`. Note that we may also explicitly initialize one or more of the enumerators. For example,

```
enum weekday {sun=1, mon, tue=2, wed, thu, fri=3, sat};
```

leads to `1,2,2,3,4,3,4` values to the enumerators `sun, mon, tue, wed, thu, fri, sat`. Since the enumerator `sun` is assigned the value `1` and the next enumerator `mon` is not assigned any value, the enumerator `mon` receives the value `2`. Similar explanations hold for the other enumerators. It is also clear that enumerators may have the same values.

Just like structures, we declare an enumerator data type above the `main()`. Also, using `typedef`, we can shorten the definition. For example, consider the following code:

```
#include <stdio.h>
typedef enum {sun=1, mon, tue=2, wed, thu, fri=3, sat} weekday;
int main()
{
 weekday d1,d2;
 d1=mon;
 d2=thu;
 printf("%d %d\n",d1,d2);
 return 0;
}
```

Listing 74: C Program "aenu.c"

The output of the code is `2 4`. We may not use a tag when we declare using `typedef`.

Just like structures, a function can have enumeration type(s) as argument(s), and the type of a function can be an enumeration too. We know that the Boolean data type represents logical states and can only hold one of two possible values: true (1) or false (0). The following program implements a function that returns a Boolean enumeration type. The program decides whether a given year is a leap year or not.

```
#include <stdio.h>
typedef enum {false,true} boolean;
boolean leap_year(int);
int main()
{
int year;
printf("Enter year:");
scanf("%d",&year);
if(year <= 0)
{
 printf("Invalid year\n");
 return 0;
}
if(leap_year(year)==true)
{
 printf("Year %d is a leap year\n",year);
}
else
{
 printf("Year %d is not a leap year\n",year);
```

```
}
return 0;
}

boolean leap_year(int year)
{
 return (boolean)(((year%4==0) && (year%100!=0)) || (year%400==0));
}
```

Listing 75: C Program "bleap.c"

Here are a few input-output.

```
Enter year:1400
Year 1400 is not a leap year
---------------------------
Enter year:1600
Year 1600 is a leap year
```

The expression `(((year%4==0) && (year%100!=0))||(year%400==0))` evaluates to zero for a non-leap year and to unity for a leap year. Though the explicit casting `(boolean)` is not necessary, it is good practice to use it. Note that the `leap_year()` is essentially an `int` function that returns either `0` or `1`. Since we have declared it to be an `enum` function, we can say that the return value is true (1) or false (0).

From the above discussion, it is clear that enumerators are useful for a data type whose range of integer values is limited. We shall use Boolean enumerators while discussing various abstract data types.

Return to Lecture Topics